RuleGrowth: Mining Sequential Rules Common to Several Sequences by Pattern-Growth

Philippe Fournier-Viger
Dept. of Computer Science and Information Engineering, National Cheng Kung University, Tainan, Taiwan, ROC
philippe.fv@gmail.com

Roger Nkambou
Dept. of Computer Science, Univ. of Quebec in Montreal
Montréal, Canada
nkambou.roger@uqam.ca

Vincent Shin-Mu Tseng
Dept. of Computer Science and Information Engineering
National Cheng Kung University
Tainan, Taiwan , ROC
tsengsm@mail.ncku.edu.tw

ABSTRACT
Mining sequential rules from large databases is an important topic in data mining fields with wide applications. Most of the relevant studies focused on finding sequential rules appearing in a single sequence of events and the mining task dealing with multiple sequences were far less explored. In this paper, we present RuleGrowth, a novel algorithm for mining sequential rules common to several sequences. Unlike other algorithms, RuleGrowth uses a pattern-growth approach for discovering sequential rules such that it can be much more efficient and scalable. We present a comparison of RuleGrowth's performance with current algorithms for three public datasets. The experimental results show that RuleGrowth clearly outperforms current algorithms for all three datasets under low support and confidence threshold and has a much better scalability.

Categories and Subject Descriptors
H.2.8 [**Information Systems**]: Database Applications – *data mining.*

General Terms
Algorithms, Performance.

Keywords
Sequential rule mining, algorithm, pattern-growth.

1. INTRODUCTION
Discovering temporal relationships between events stored in large databases is important in many domains, as it provides a better understanding of the relations between events, and sets a basis for the prediction of events. For example, in international trade, one could be interested in discovering relations between the appreciation of currencies to make trade decisions. Various methods have been proposed for mining temporal relations between events in databases (see for example [12], for a survey). In the field of data mining, one of the most popular set of techniques for discovering temporal relations between events in discrete time series is sequential pattern mining [2], which consists of finding sequences of events that appear frequently in a sequence database. However, knowing that a sequence of events appear frequently in a database is not sufficient for the prediction of events. For example, it is possible that some event *c* appears frequently after some events *a* and *b* but that there are also many cases where *a* and *b* are not followed by *c*. In this case, predicting that *c* will occur if *a* and *b* occur on the basis of a sequential pattern *abc* could be a huge mistake. Thus, for prediction, it is desirable to have patterns that indicate how many times *c* appeared before *ab* and how many times *ab* appeared and *c* did not. But adding this information to sequential patterns cannot be done easily as sequential patterns are lists of events that can contain several events – not just two, as in the previous example– and current algorithms have just not been designed for this.

The alternative to sequential pattern mining that addresses the problem of prediction is sequential rule mining [4, 5, 8, 9, 11, 13]. A sequential rule (also called episode rule, temporal rule or prediction rule) indicates that if some event(s) occurred, some other event(s) are also likely to occur with a given confidence or probability. Sequential rule mining has been applied in several domains such as stock market analysis [4, 11], weather observation [8], drought management [5, 9] and e-learning [6].

Several algorithms for sequential rule mining have been developed (e.g. [4, 5, 8, 9, 10, 11, 13, 14]). They can be grouped into two main categories. The first one is algorithms for mining sequential rules appearing in a single sequence of events. The most famous approach from this group is that of Mannila et al [13]. Among several applications, it was used for the analysis of alarm flow in telecommunication networks. It consists of finding rules respecting a minimal confidence and support. These rules are of the form X⇒Y, where X and Y are two sets of events, and are interpreted as "if event(s) X appears, event(s) Y are likely to occur with a given confidence afterward". Other algorithms that have a similar aim are the ones of Hamilton & Karimi [8], Hsieh et al [11] and Deogun & Jiang [5].

The second category consists of the algorithms that discover rules in sets of sequences (sequence databases). This category can be further divided into two subcategories. First, there are algorithms that discover rules appearing frequently in sequences, no matter if the rules appear in a single sequence or in several sequences. An example of such algorithm is the one of Harms et al. [9]. It discovers rules such that the right part occurs frequently after the left part within user-defined time windows.

SAC'11, March 21-25, 2011, TaiChung, Taiwan.

The second subcategory consists of algorithms for finding rules that are common to many sequences. Unlike the other subcategory, the goal of these algorithms is not to discover rules that appear frequently in sequences. Instead it is to find rules that are common to several sequences. Examples of applications of these algorithms are an intelligent tutoring agent named CTS that was integrated in an e-learning system [6]. The tutoring agent records its interactions with each learner as a sequence and then finds rules common to all sequences. The agent then uses these rules to make prediction about the outcome of interactions with next learners, to adapt its behavior [6]. A second example is an agent that discovers patterns in the demonstrations of a task, to learn the task, to then provide explanations to users about the task [16].

Up to now, we know only three algorithms that have been designed for the task of mining sequential rules common to several sequences as mentioned above. The first algorithm, by Lo et al. [14], discovers rules where the left and right parts of each rule are sequences of itemsets (sets of items). One particularity of this type of rules is that items of the left part or the right part of a rule have to appear with exactly the same ordering in a sequence so that it is counted as an occurrence of the rule (otherwise it will be counted as an occurrence of a distinct rule), even for the simplest case where the left or right part are a single itemset. Recently, we have defined a more general form of sequential rules common to several sequences. We defined rules such that items from the left or right parts do not always need to be ordered in the same way in sequences, as long as the items of the left part occur before items of the right part. This kind of rules is not discovered by the algorithm of Lo et al. In previous works, we proposed two algorithms for discovering these rules, which are named CMRules and CMDeo [10]. In this paper, we propose a novel algorithm that outperforms them.

The rest of this paper is organized as follows. The next section present the problem of mining sequential rules common to several sequences as we have defined it and present a brief outline of the CMRules and CMDeo algorithms. The third section presents the RuleGrowth algorithm. The fourth section presents an evaluation of its performance on real datasets. Finally, the last section draws a conclusion.

2. PROBLEM DEFINITION AND CURRENT ALGORITHMS

We defined the problem of mining sequential rules common to several sequences as follow [10]. A *sequence database* SD is a set of sequences $S=\{s_1, s_2 \ldots s_n\}$ and a set of items $I=\{i_1, i_2, \ldots i_n\}$, where each sequence s_x is an ordered list of itemsets $s_x=\{X_1, X_2, \ldots X_n\}$ such that $X_1, X_2, \ldots X_n \subseteq I$. Note that this definition of a sequence database is the one that is used for sequential pattern mining [2].

A *sequential rule* $X \Rightarrow Y$ is defined as a relationship between two itemsets X, $Y \subseteq I$ such that $X \cap Y = \emptyset$ and X, Y are not empty. The interpretation of a rule $X \Rightarrow Y$ is that if the items of X occur in some itemsets of a sequence, the items in Y will occur in some itemsets afterward in the same sequence. Note that there is no ordering restriction between items in X and between items in Y, and that item from X and items from Y do not need to appear in a same itemset in a sequence. Two interestingness measures are defined for a sequential rule, which are an adaptation of the measures typically used in sequential rule mining [4, 9, 13] and are similar to those used in association rule mining [1]. The first measure is the rule's *sequential support* and is defined as: $sup(X \Rightarrow Y) = sup(X \blacksquare Y) / |S|$. The second measure is the rule's *sequential confidence* and is defined as: $conf(X \Rightarrow Y) = sup(X \blacksquare Y) / sup(X)$. Here, the notation $sup(X \blacksquare Y)$ denotes the number of sequences from a sequence database where all the items of X appear before all the items of Y. Formally, an item *x* is said to occur before another item *y* in a sequence $s_x=\{X_1, X_2, \ldots X_n\}$ if there exists integers $k < m$ such that $x \in X_k$ and $y \in X_m$. The notation sup(X) for an itemset X represents the number of sequences that contains the items of X.

The *problem of mining sequential rules common to several sequences* is to find all sequential rules from a sequence database such that their support and confidence are respectively higher or equal to some user-defined thresholds *minSup* and *minConf*. Those rules are said to be the "valid rules". As an example, figure 1 shows a sequence database containing four transactions and some valid rules found for *minSup* = 0.5 and *minConf* = 0.5. In each sequence of the database, each single letter represent an item and item(s) between parentheses represent an itemset.

ID	Sequences
1	(a b), (c), (f), (g), (e)
2	(a d), (c), (b), (e f)
3	(a), (b), (f) (e)
4	(b), (f g)

→

ID	Rule	Support	Confidence
1	a b c ⇒ e	0.5	1.0
2	a ⇒ c e f	0.5	0.66
3	a b ⇒ e f	0.5	1.0
4	b ⇒ e f	0.75	0.75
5	a ⇒ e f	0.75	1.0
6	c ⇒ f	0.5	1.0
7	a ⇒ b	0.5	0.66
	…	…	…

Figure 1. A sequence database and some sequential rule found

The first algorithm that we proposed for this problem is CMRules [10]. The strategy of this algorithm is to prune the search space for sequential rules by first discovering items that occur jointly in many sequences. The algorithm proceeds as follows. First it removes the temporal information from the sequence database taken as input to get a transaction database. Then it applies an association rule mining algorithm [1] on the transaction database to generate all association rules such that their support and confidence is at least *minSup* and *minConf*. Finally, for each association rule found, the algorithm scans the original database to eliminate rules that do not meet minimum confidence and support thresholds according to the time ordering. [10]. The main benefits of CMRules it is easy to implement because existing association rule mining algorithms can be partly reused and that it performs better than CMDeo for some datasets [10]. Its main weakness is that its performance depends on the number of association rule found. If this set is large, CMRules becomes inefficient [10].

The second algorithm is CMDeo [10], which is an adaptation of the Deogun et Jiang's algorithm [5]. CMDeo finds all items that appear in at least *minSup* sequences by scanning the sequence database one time. Then, for each pair of such items *x,y*, the algorithm generate candidates rules $\{x\} \Rightarrow \{y\}$ and $\{y\} \Rightarrow \{x\}$. The sequential support and confidence of each rule is then calculated by scanning the database to see if they respect *minSup* and *minConf*. Once, these rules are found, the algorithm then recursively find larger candidate rules by combining rules of smaller size in a level-wise manner (similar to Apriori [1]). This is

done by two separate processes. Left-side expansion is the process of taking two candidate rules X⇒Y and Z⇒Y, where X and Z are itemsets of size *n* sharing *n*-1 items, to generate a new larger candidate rule X∪Z ⇒Y. Right-side expansion is the process of taking two candidate rules Y⇒X and Y⇒Z, where X and Z are itemsets of size n sharing *n-1* items, to generate a new larger candidate rule Y⇒X∪Z. After candidate rules are generated by left/right side expansion their sequential support and confidence are calculated by scanning sequences. To prune the search space for candidate rules, it can be shown easily that expanding the left side of a rule not respecting *minSup* will not result in valid sequential rules, and that expanding the right side of a rule not respecting *minSup* or *minConf* will not generate valid sequential rules [10]. CMDeo performs considerably better than CMRules for some datasets [10]. However, for some datasets, the search space is such that CMDeo will generate a very large number of candidate rules that are not valid rules, which can makes it inefficient.

3. THE RULEGROWTH ALGORITHM

The algorithm that we propose uses an approach that is different from CMDeo and CMRules. Instead of using a generate-candidate-and-test approach, it relies on a Pattern-Growth approach similar to the one used in the PrefixSpan [7] algorithm for sequential pattern mining. RuleGrowth first find rules between two items and then recursively grow them by scanning the database for single items that could expand their left or right parts (these processes are called left and right expansions). Like PrefixSpan, RuleGrowth also includes some ideas to prevent scanning the whole database every time.

3.1 Observations

Before presenting the algorithm, we present the observations and some ideas that were used to develop the algorithm. In the following we say that a rule X⇒Y is of size **k*m** if |X| =k and |Y|=m. First, we present four lemmas:

Lemma 1: If an item *i* is added to the left side of a rule r:X⇒Y, the support of the resulting rule r' can be lower or equal to the support of r. **Proof:** The support of r and r' are respectively sup(X■Y)/|S| and sup(X∪{i}■Y)/|S|. Since sup(X■Y)≥ sup(X∪{i}■Y), sup(r) ≥ sup(r').

Lemma 2: If an item *i* is added to the right side of a rule r:X⇒Y, the support of the resulting rule r' can be lower or equal to the support of r. **Proof:** The support of r and r' are respectively sup(X■Y)/|S| and sup(X■Y∪{i})/|S|. Since sup(X■Y)≥ sup(X■Y∪{i}), sup(r) ≥ sup(r').

Lemma 3: If an item *i* is added to the left side of a rule r:X⇒Y, the confidence of the resulting rule r' can be lower, higher or equal to the confidence of r. **Proof:** The confidence of r and r' are respectively sup(X■Y)/|sup(x)| and sup(X∪{i}■Y)/sup(X∪{i}). Because sup(X■Y) ≥ sup(X∪{i}■Y) and sup(X)≥ sup(X∪{i}), conf(r) can be lower, higher or equal to conf(r')

Lemma 4: If an item *i* is added to the right side of a rule r:X⇒Y, the confidence of the resulting rule r' can be lower or equal to the confidence of r. **Proof:** The confidence of r and r' are respectively sup(X■Y)/|sup(X)| and sup(X■Y∪{i})/sup(X). Since sup(X■Y) ≥ sup(X■Y∪{i}), conf(r) ≥ conf(r').

The idea of RuleGrowth is to grow rule by starting with rules of size 1*1 and to recursively add one item at a time to the left or right side of a rule (left/right expansions) to find larger rules. The next paragraphs present four problems that we have faced to develop RuleGrowth and the solutions that we have found to overcame them.

Problem 1: If we grow rules by performing left/right expansions, some rules can be found by different combinations of left/right expansions. For example, consider the rule {a,b}⇒{c,d}. By performing, a left and then a right expansion of {a}⇒{c}, one can obtain the rule {a,b}⇒{c,d}. But this rule can also be obtained by performing a right and then a left expansion from {a}⇒{c}.

Solution 1: A simple solution to this problem is to not allow performing a right expansion after a left expansion but to allow performing a left expansion after a right expansion. Note that an alternative solution is to not allow a left expansion after a right expansion.

Problem 2: A second problem is that many rules may need to be expanded to find all valid rules. Therefore it would be desirable to be able to identify rules that are guaranteed to not generate valid rules if they are expanded.

Solution 2: To address this problem, we use the previous lemmas. In particular, lemma 1 and 2 implies that expanding a rule r such that sup(r)<minsup will not result in a valid rule. Therefore, all such rules should not be expanded. However, we cannot apply a similar pruning for confidence because of Lemma 3.

Problem 3: Another problem is that some rules can be found several times by performing left/right expansions with different items. For example, consider the rule {bc}⇒{d}. A left expansion of {b}⇒{d} with item *c* can results in the rule {bc}⇒{d}. But that latter rule can also be found by performing a left expansion of {c}⇒{d} with *b*.

Solution 3: To solve this problem, we chose to only add an item to an itemset of a rule if the item is greater than each item in the itemset according to the lexicographic ordering. In the previous example, this would mean that item *c* would be added to the left itemset of {b}⇒{d}. But *b* would not be added to the left itemset of {c}⇒{d}.

Problem 4: Another challenge is to have an efficient way of determining each item that can expand a rule to generate a valid rule.

Solution 4: Our solution is to scan the sequences containing each rule to count the support of items that could expand the rule. If an item appears in at least *minsup* of these sequences, this means that adding the items to the rule would result in a rule having at least the minimal support. To make this process more efficient, we have made the two following observations. First, for a rule, it can be easily shown that any item that could expand the left itemset of a rule must appear before the last occurrence of its right itemset in at least *minSup**|S| sequences that contains the rule. Similarly, it can be easily shown that any item that could expand the right itemset of a rule has to appear after the first occurrence of its left itemset in at least *minSup**|S| sequences containing the rule. In RuleGrowth, we use these two properties to avoid scanning sequences completely when possible. To do this optimization, RuleGrowth keeps track for each sequence of the first and/or last

occurrence of a rule's itemsets, and this information is updated after each recursion. This will be explained in detail in the next subsection.

3.2 The Algorithm

We now present the RuleGrowth algorithm. Its main procedure is named RULEGROWTH and is shown on Figure 2. It takes as parameters a sequence database and the *minsup* and *minconf* thresholds. This procedure first generates all rules r of size 1*1 such that sup(r) ≥ *minsup* and then call two recursive procedures for growing each rule. These procedures, named EXPANDLEFT and EXPANDRIGHT, are presented on figure 3 and 4.

RULEGROWTH(*database, minsup, minconf*)

1. Scan the database one time. During this scan, for each item *c*, record the *sids* of the sequences that contains *c* in a variable *sids_c* and the position of the first and last occurrence of *c* for each *sid* in hash tables respectively named *firstOccurences_c* and *lastOccurences_c*.
2. FOR each pairs of items *i, j* such that |*sids_i*|≥ *minsup* and | *sids_j*|≥ *minsup:*
3. *sidsi■j* :={}.
4. *sidsj■i* :={}.
5. FOR each sid *s* such that *s* ∈ *sids_i* and *s* ∈ *sids_j*
6. IF *firstOccurences_i(s)* is before *lastOccurences_j(s)*, THEN s*idsi■j* := s*idsi■j* ∪{*s*}.
7. IF *firstOccurences_j(s)* is before *lastOccurences_i(s)*, THEN s*idsj■*i := s*idsj■i* ∪{*s*}.
8. END FOR
9. IF (|*sidsi■j*| / |*database*|) ≥ *minsup* THEN
10. EXPANDLEFT({i}⇒{j}, *sids_i, sidsi■j, lastOccurences_j*).
11. EXPANDRIGHT({i}⇒{j}, *sids_i, sids_j, sidsi■j, firstOccurences_i, lastOccurences_j*).
12. IF (|*sidsi■j*| / |*sids_i*|) ≥ *minconf* THEN OUTPUT rule {i}⇒{j} with its confidence and support.
13. END IF
14. … [lines 9 to 13 are repeated here with *i* and *j* swapped]…
15. END FOR

Figure 2. The RuleGrowth algorithm

The first step of the RULEGROWTH procedure is to scan the database one time to count the support of each item. The algorithm reads each sequence from the beginning to the end, and notes for each item *c*, the sid (sequence id) of each sequence that contains *c* in a variable *sids_c*, and the first and last occurrence of the item for each of those *sids* in variables respectively named *firstOccurences_c* and *lastOccurences_c*.

Given this information, all rules of size 1*1 are generated very efficiently without scanning the database again. This is done as follows. The algorithms takes each pair of items *i, j*, where *i* and *j* each have at least *minsup**|*S*| sids (if this condition is not met, it means that no rule having at least the minimal support could be created with *i* and *j*). The algorithm then initialize two variables *sidsi■j*={} and *sidsj■i*={} for counting the support of the rules {i}⇒{j} and {j}⇒{i}. Then, the algorithm loops over each sid containing *i* and *j* to check for each sid if the first occurrence of *i* occurs before the last occurrence of *j* (this check is very fast since the first and last occurrence of each item for each sequence has been recorded). If it is the case, the current sid is added to *sidsi■j*. Similarly, if first occurrence of *j* is before the last occurrence of *i*, the current sid is added to *sidsj■i*. After this loop, the support of the rule {i}⇒{j} is simply obtained by dividing *sidsi■j* by |S|. If the support is higher or equal to *minsup*, the procedure EXPANDLEFT and EXPANDRIGHT are called to try to expand the rule's left and right parts and the confidence of the rule is calculated by dividing *sidsi■j* by |*sids_i*|. If the confidence is higher or equal to *minconf*, the algorithm output the rule. After this, the same process is repeated for the rule {j}⇒{i}.

The procedure EXPANDLEFT (cf. fig. 3) tries to expand the left side of a rule. Its takes as parameters a rule I⇒J to be expanded (*ruleIJ*), the list of sids containing I (*sidsI)*, the list of sids containing I followed by J (*sidsI■J)* and a structure (an hashmap in our implementation) indicating the last occurrence of J for each sid (*lastOccurences_J)*. The procedure first tries to find items that could expand the rule. It does this by looping over each sequence from *sidsI■J*. For each item *c* appearing before the last occurrence of J in a sequence, the sid of the sequence is added to a variable *sidsIc■J*. Note that here only items lexically larger than those already in I are considered (cf. Solution 3). After this loop, for each item found, the support of the rule I∪{c}⇒J is calculated by dividing |*sidsIc■J*| by |*S*|. If the support is higher or equal to *minsup*, the procedure EXPANDLEFT is called to check if the left itemset of the rule could be further expanded (cf. Solution 1). The EXPANDLEFT procedure is called with the list of sids containing I∪{c} (*sidsIc*) as second parameter instead of *sidsI*. *sidsIc* is obtained by doing a simple loop over *sidsI* to check if each sid is an element of *sids_c*. After calling EXPANDLEFT, the procedure checks the confidence of I∪{c}⇒J. It is calculated by dividing |*sidsIc■J*| by |*sidsI* |. If the confidence is higher or equal to *minconf*, the procedure output the rule (the rule is valid).

EXPANDLEFT(*ruleIJ, sidsI, sidsI■J, lastOccurences_J*)

1. FOR each *sid* ∈ *sidsI■J*, scan the sequence *sid* from the first itemset to the itemset before the last occurrence of J. For each item *c* appearing in these sequences that is lexically larger than items in I, record in a variable *sidsIc■J* the *sids* of the sequences where *c* is found.
2. END FOR
3. FOR each item *c* such that (*sidsIc■J* / |*S*|) ≥ *minsup* :
4. *sidsIc* = {}.
5. FOR each *sid* ∈ *sidsI*
6. IF(*sid* ∈ *sids_c*) THEN *sidsIc* := *sidsIc* ∪{sid}.
7. END FOR
8. EXPANDLEFT(I∪{c}⇒J, sidsIc, sidsIc■J, *lastOccurences_j*).
9. IF (|*sidsIc■J*| / |*sidsI* |)≥ *minconf* THEN OUTPUT rule I∪{c}⇒J.
10. END FOR

Figure 3. The EXPANDLEFT procedure

The procedure EXPANDRIGHT (cf. fig. 4) is very similar to EXPANDLEFT. But, it performs a few more operations and takes more parameters because it calls EXPANDLEFT and EXPANDRIGHT (cf. Solution 1). The parameters are a rule I⇒J to be expanded (*ruleIJ*), the list of sids containing I (*sidsI)*, the list of sids containing J (*sidsJ*), the list of sids containing I followed by J (*sidsI■J)*, a structure (an hashmap in our implementation) indicating the first occurrence of I for each sid (first*Occurences_I)*

and a structure indicating the last occurrence of J for each sid (*lastOccurences_J)*. The procedure first tries to find items that could expand the right side of the rule. It does this by looping over each sequence from *sidsI■J*. For each item *c* appearing after the first occurrence of J in a sequence, the sid of the sequence is added to a variable *sidsIc■J*. Note that here only items lexically larger than those already in J are considered (cf. Solution 3). After this loop, for each item found, the support of the rule I⇒JU{c} is calculated by dividing |*sidsIc■J*| by |*S*|. If the support is higher or equal to *minsup*, the procedure EXPANDLEFT and EXPANDRIGHT are called to check if the left itemset of the rule could be further expanded by left or right expansions. EXPANDLEFT is called with the rule I⇒JU{c}, the list of sids containing I (*sidsI*), the list of sid containing I⇒JU{c} (*sidsI■Jc*) and the last occurences of JU{c} for each sequence containing I⇒JU{c} (*lastOccurence_Jc*). The parameters *sidsI■Jc* and *lastOccurence_Jc* are obtained by looping over the sequence containing J (lines 6 to 10 of the procedure) and noting each sequence containing *c* in *sidsI■Jc* and updating the position of the last occurrence of J by considering *c*. EXPANDRIGHT also takes these two updated parameters. After calling EXPANDLEFT and EXPANDRIGHT, the procedure checks the confidence of I⇒JU{c} by dividing |*sidsIc■J*| by |*sidsI*|. If it is higher or equal to *minconf*, the procedure output the rule (the rule is valid).

EXPANDRIGHT(*ruleIJ, sidsI, sidsJ, sidsI■J, firstOccurences_I, lastOccurences_J*)

1. FOR each *sid* ∈ *sidsI■J*, scan the sequence *sid* from the itemset after the first occurrence of I to the last itemset. For each item *c* appearing in these sequences that is lexically larger than items in J, record in a variable *sidsI■Jc* the *sids* of the sequences where *c* is found.
2. END FOR
3. FOR each item *c* such that |*sidsI■Jc*| ≥ *minsup**|*S*| :
4. *sidsJc* = {}.
5. *lastOccurences_Jc* := *lastOccurences_J*.
6. FOR each *sid* ∈ *sidsJ*
7. IF(*sid* ∈ *c.sids*) THEN
8. *sidsJc* := *sidsJc* ∪ {sid}.
9. Update(*lastOccurences_Jc, lastOccurences_J*, c, sid).
10. END IF
11. END FOR
12. EXPANDLEFT(I⇒JU{c}, *sidsI, sidsI■Jc, lastOccurence_Jc*).
13. EXPANDRIGHT(I⇒JU{c}, *sidsI, sidsJC, sidsI■Jc, firstOccurences_I, lastOccurences_Jc*).
14. IF (|*sidsI■Jc*| / | *sidsI* |) ≥ *minconf* THEN
15. OUTPUT rule I⇒JU{c}.
16. ENDIF
17. END FOR

Figure 4. The EXPANDRIGHT procedure

4. PERFORMANCE EVALUATION

We have implemented RuleGrowth, CMDeo and CMRules in the Java programming language. Our implementations can be downloaded from: http://www.philippe-fournier-viger.com/spmf/. We describe next a performance comparison of these three algorithms on three real-life datasets having different characteristics and representing three real-life situations.

The first dataset is Kosarak, a dataset available from http://fimi.cs.helsinki.fi/data/, which contains 990 000 click-stream data from the logs of an online news portal. For this comparison, as in [10], we used only the 70 000 first sequences of Kosarak to make the experiment faster. These sequences have an average of 7.97 items (σ =21.14) from 21144 different items. The second dataset is BMS-Webview1, a sequence database containing several months of click-stream from an e-commerce. It was used for the KDD-Cup 2001 competition and can be downloaded from http://www.ecn.purdue.edu/KDDCUP/. BMS-Webview1 differs from Kosarak principally in that it contains shorter sequences and that the set of different items is much smaller than for Kosarak. The average length of sequences is 2.51 items σ =4.85) and there is totally 497 different items.

The third dataset is Toxin-Snake [15], a sequence database from the domain of biology. It contains 192 protein sequences. For our experiment, only sequences containing more than 50 items have been kept. Keeping only these sequences has been done to make the dataset more uniform, by removing some very short sequences. This resulted in 163 long sequences containing an average of 60.61 items. The particularity of Toxin-Snake is that it is a very dense dataset: each item occurs in almost every sequence (there is on average 17.84 different items in each sequence, and Snake only contains 20 different items) and the items appearing in a sequence appears 3.39 times on average (σ =2.24).

For Kosarak, we ran the three algorithms with *minconf*=0.3 and *minsup*= 0.004, 0.00375 … 0.001. Performance of each algorithm and the number of rules found are illustrated on Figure 5.A. For this experiment a time limit of 10 000 seconds was set and a memory usage of 1 GB. Because of these limits, CMDeo and CMRules were unable to provide results for support lower than 0.0025 (12 994 rules) and 0.00175 (100 492 rules) respectively, while RuleGrowth was run until 0.001 (2,655,064 rules).

For BMS-Webview1, the algorithms were run for *minconf*=0.2 and *minsup*= 0.00085, 0.000825 … 0.0006. Performance of each algorithm and the number of rules found are illustrated on Figure 5.B. For this experiment, a time limit of 2500 seconds was set and a memory usage of 1 GB. As for the previous experiment, RuleGrowth outperformed CMDeo and CMRules.

For the Snake dataset, algorithms were ran for *minconf*=0.2 and *minsup*= 0.96, 0.94 … 0.7. Performance of each algorithm and the number of rules found are illustrated on Figure 5.B. For this experiment a time limit of 700 seconds was set and a memory usage of 1 GB. For this experiment, RuleGrowth also shown a better performance than CMRules and CMDeo.

From these three experiments, we can conclude that RuleGrowth is much more efficient than CMRules and CMDeo and has a much better scalability. Although we did not measure the memory usage in these experiments, only CMDeo and CMRules exceeded the memory limit. The reason why RuleGrowth does not use a lot of memory is that it relies on the Pattern-Growth approach instead of a generate-candidate-and-test approach that is used by CMRules and CMDeo that can involve generating and maintaining large sets of candidates.

5. CONCLUSION

In this paper, we presented RuleGrowth, a novel algorithm for mining sequential rules common to several sequences. Unlike

previous algorithms, it does not use a generate-candidate-and-test approach. Instead, it uses a pattern-growth approach for discovering valid rules such that it can be much more efficient and scalable. It first finds rules between two items and then recursively grows them by scanning the database for single items that could expand their left or right parts. We have evaluated the performance of RuleGrowth by comparing it with the CMDeo and CMRules algorithms. Results show that RuleGrowth clearly outperforms CMRules and CMDeo and has a better scalability.

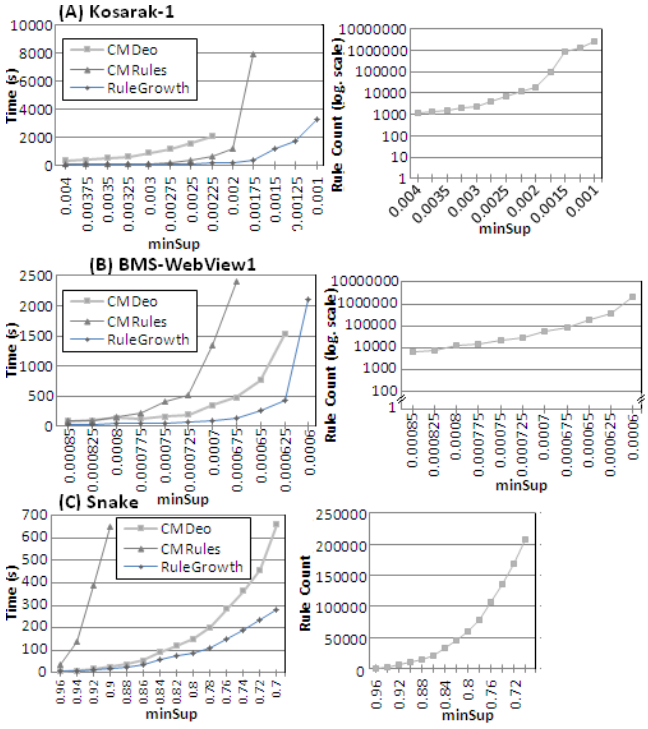

Figure 5. Results of the experiments

The RuleGrowth algorithm can be extended in several ways. For example, one simple extension is to allow specifying size constraints on the size of the left and right parts of rules to be found. This type of constraints can be easily enforced since rules are created one item at a time. Another possible extension that would be easy to implement is to allow specifying constraint on items that can, should or should not appear in the left or the right part of a rule. This would allow users to refine their search for rules. For future works, we will explore further optimizations that could improve the algorithm performance and we will also try alternative approaches.

6. ACKNOWLEDGMENTS

The authors thank the Fonds Québécois de la Recherche sur la Nature et les Technologies (FQRNT) for its financial support.

7. REFERENCES

[1] Agrawal, R., Imielminski, T., and Swami, A. Mining Association Rules Between Sets of Items in Large Databases, In *Proc. SIGMOD Conference*, (Washington D.C., USA, May 26-28, 1993) 207-216.

[2] Agrawal, R. and Srikant, R. Mining Sequential Patterns. In *Proc. Int. Conf. on Data Engineering* (Taipei, Taiwan, March 6-10, 1995), 3-14.

[3] Cheung, D.W., Han, J., Ng. V. and Wong., Y. Maintenance of discovered association rules in large databases: An incremental updating technique. In *Proc. ICDE 1996* (New Orleans, USA, February 26 - March 1, 1996), 106-114.

[4] Das., G., Lin, K.-I., Mannila, H., Renganathan, G., and Smyth, P. Rule Discovery from Time Series. In *Proc. 4th Int. Conf. on Knowledge Discovery and Data Mining* (New York, USA, August 27-31, 1998), 16-22.

[5] Deogun, J.S. & Jiang, L. Prediction Mining – An Approach to Mining Association Rules for Prediction. In *Proc. of RSFDGrC 2005 Conference* (Regina, Canada, August 31st-September 3rd, 2005), 98-108.

[6] Faghihi, U., Fournier-Viger, P., Nkambou, R. and Poirier, P. Generic Episodic Learning Model Implemented in a Cognitive Agent by Means of Temporal Pattern Mining. In *Proceedings of IEA-AIE 2010* (Cordoba, Spain, June 1-4 2010), 438-449.

[7] Pei, J., Han, J. et al.: Mining Sequential Patterns by Pattern-Growth: The PrefixSpan Approach. *IEEE Trans. Knowledge and Data Engineering*, 16, 10 (2004), 1-17.

[8] Hamilton, H. J. and Karimi, K. The TIMERS II Algorithm for the Discovery of Causality. In *Proc. 9th Pacific-Asia Conference on Knowledge Discovery and Data Mining* (Hanoi, Vietnam, May 18-20, 2005), 744-750.

[9] Harms, S. K., Deogun, J. and Tadesse, T. 2002. Discovering Sequential Association Rules with Constraints and Time Lags in Multiple Sequences. In *Proc. 13th Int. Symp. on Methodologies for Intelligent Systems* (Lyon, France, June 27-29, 2002), pp. .373-376.

[10] Fournier-Viger, P., Faghihi, U., Nkambou, R. and Mephu Nguifo, E. CMRules: An Efficient Algorithm for Mining Sequential Rules Common to Several Sequences. In *Proceedings 23th Intern. Florida Artificial Intelligence Research Society Conference* (Daytona, USA, May 19-21, 2010), AAAI Press, 410-415.

[11] Hsieh, Y. L., Yang, D.-L. and Wu, J. Using Data Mining to Study Upstream and Downstream Causal Relationship in Stock Market. In *Proc. 2006 Joint Conference on Information Sciences* (Kaoshiung, Taiwan, October 8-11, 2006).

[12] Laxman, S. and Sastry, P. A survey of temporal data mining. *Sadhana* 3 (2006), 173-198.

[13] Mannila, H., Toivonen and H., Verkano, A.I. Discovery of frequent episodes in event sequences. *Data Mining and Knowledge Discovery*, 1, 1 (1997), 259-289.

[14] Lo, D., Khoo, S.-C., Wong, L.. Non-redundant sequential rules - Theory and algorithm. *Information Systems*, 34, 4-5 (2009), 438-453.

[15] Agrawal, R. and Shafer, J. C. Parallel Mining of Association Rules. *IEEE Transaction on Knowledge and Data Engineering*, 8, 6 (1996), 962-969.

[16] Fournier-Viger, P. 2010. *Knowledge Discovery in Problem-Solving Learning Activities*. Ph.D. thesis. University of Quebec in Montreal